

IWM Scheduling: A Knowledge-Based Partitioning Approach

April 19, 1999

Introduction

This document describes a new approach to IWM Scheduling¹, based on partitioning the problem and then applying distinct algorithms to each partition. The algorithm and parameters to be applied to each partition are determined, in version -1, by a forward-chaining rule-based system implemented in MasterLink's Tiny Rule Engine (TRE). My document "Introduction to the MasterLink Tiny Rule Engine" provides a brief description of TRE.

We begin with a high level description of the IWM Scheduling problem, followed by an introduction to the partitioning approach. The version -1 scheduling problem is discussed in detail, followed by a brief introduction to the version 0.0 and version 1.0 scheduling problems.

This is followed by a description of several types of algorithms that may be used to schedule work for individual partitions, including an exhaustive algorithm used in low complexity partitions, probabilistic algorithms, and work-schedule centric and job-centric deterministic algorithms. Each type of algorithm provides tradeoffs in time/space/optimality of answer.

Scheduling for a partition consists of running one of the above algorithms, followed by an optional optimization pass. This optimization pass considers jobs in the Unassigned Job Queue and attempts to assign high priority jobs in the queue to work schedules, thus improving the quality of the work schedules. This technique, called rescheduling is then described, including its relationship to scheduling unplanned jobs. Finally, the future of partitioning approaches to IWM scheduling is briefly considered.

The place of the IWM scheduling algorithms in the MasterLink Agent framework is not discussed in this document. A brief introduction to the MasterLink Agents included in version -1 may be found in my document "MasterLink IWM Agents: Basic Definition".

The IWM Scheduling Problem

Introduction

This section briefly describes the IWM scheduling problem, both as it is originally presented in V -1 and as it will be defined for V 1.0. These two scheduling problems differ in size (number of jobs, number of workers, complexity of the object model) and scope (emphasis on hard constraints vs. emphasis on soft constraints).

Scheduling considers the assignment of jobs, including their tasks, to workers, that is, to the worker's work schedule. There are tasks to be scheduled; these tasks are grouped into jobs that must be assigned to workers. Each worker has a schedule. Scheduling is the process of assigning jobs to work schedules to maximize the utility of the overall set of work schedules.

Sizing the Version -1 Scheduling Problem

Size: The size of the problems encountered in the V -1 prototype is as follows:

- ❖ 2 locations, each of which have 2 buildings

¹ IWM Scheduling is the scheduling of workers in a job-based work management environment.

- ❖ 30 Workers, including electricians, plumbers, A/C technicians and maintenance technicians, located as follows:

| | Electricians | Plumbers | A/C Techs | Maint Techs | Total |
|-------------------|--------------|----------|-----------|-------------|-------|
| Location 1 Bldg A | 2 | 2 | 2 | 3 | 9 |
| Location 1 Bldg B | 2 | 1 | 3 | 3 | 9 |
| Location 2 Bldg Y | 2 | 1 | 2 | 1 | 6 |
| Location 2 Bldg Z | 1 | 1 | 2 | 2 | 6 |
| Total | 7 | 5 | 9 | 9 | 30 |

- ❖ 100 Targets
- ❖ Job Priorities are expressed as integers, range 1 (lowest) to 10 (highest)
- ❖ Tasks Estimated Time ranges from 15 minutes to 90 minutes
- ❖ Average job consists of 5 tasks
- ❖ A total of 200 jobs are schedulable

Ontology: The vocabulary for the problems encountered in the version -1 demonstration consists of attributes of the schedulable jobs, their tasks and target, and available workers and their work schedules:

- ❖ Jobs: Priority, State (created, pending Assigned, assigned, dispatched)
- ❖ Tasks: Estimated time, Labor Type
- ❖ Targets: Location, Building
- ❖ Workers: Skill, Location, Building
- ❖ Work Schedule: Shift Start, Shift End (to compute shift length)

A job is a set of one or more tasks for the same target and skill that are to be put on the same work schedule. Jobs are categorized by the location and building they are performed at, the total time the job will take (the sum of the elapsed time for the tasks on the job), the skill required for the job, and the job's priority.

Workers are categorized by the location and building they cover, the skills they provide, and their work schedule, which consists of jobs assigned to that worker for the day. We only consider the total amount of time for all the jobs assigned to the worker; that total must be not greater than the shift length for that worker.

Hard constraints:

- ❖ A worker can only work on one job at a time.
- ❖ Compute the time a job requires by summing the time each of the job's task requires.
- ❖ The total amount of time per shift is a constant, defined by policy (for the demo, a constant, 8 hrs)
- ❖ Workers must be "compatible" with a job, which for the demo is defined as:
 - ❖ Workers must have a skill which matches the skill required for the job, and
 - ❖ Workers must be at a location and building appropriate for the job:
 - ❖ The Worker location must match the job location
 - ❖ Depending on policy about workers (worker-works-all-buildings-at-location?) the worker building must match the job building

Soft Constraints:

Soft constraints are based on maximizing the "fit" of the jobs to the work schedules. The fit is a numeric rating of the goodness of a work schedule: a higher rating indicates a better fit of a set of jobs to a worker's work schedule.

We first look at this from the viewpoint of a fixed set of work schedules where the task is to place schedulable jobs onto the work schedules, subject to the constraints. From this viewpoint, the soft constraints include:

- ❖ Given two jobs appropriate for a worker, choose the one with higher priority.
- ❖ Given two jobs appropriate for a worker, both of which have the same priority, choose the longer one.

To automate this process requires a fitness function (see below) with the following characteristics:

- ❖ Priority is more important than elapsed time. This is required for the first bullet above.
- ❖ Reward high priority jobs. The best schedule, everything else being equal, is composed of all high priority jobs. This enforces the first bullet above.
- ❖ Penalize holes in a work schedule. The best schedule, everything else being equal, takes up the total shift time. This enforces the second bullet above.

Another way to look at this is as a fixed set of workers, and a set of jobs that are assigned to them. From this viewpoint, the soft constraints include:

- ❖ Given a job appropriate for two workers, assign the job to the worker that has the smaller amount of available time. That is, fill small time-slots first. The rationale is that larger time slots are easier to fill.
- ❖ Given two jobs appropriate for a workers, assign the job to the worker that has the greater priority.

To automate this process requires choosing a fitness function with the following characteristics:

- ❖ Penalize larger holes in a work scheduler more than smaller holes. This enforces the first bullet above.
- ❖ Reward high priority jobs. This enforces the second bullet.

The Version 1.0 Scheduling Problem

Version 1.0 presents a more complex object model, richer in soft constraints. Many of the hard constraints used in version -1 represent simplifications of aspects of the problem that are represented in V 1.0 by soft constraints.

Example: Skill. In V -1 the skill required by the job must match the skill of the worker. In V 1.0, the notion of skill is enhanced (electricians come in various flavors, based on experience and knowledge). The relationship of worker skill to job skill is then phrased as a set of soft constraints that capture the business policy with respect to the flavors or electricians and the flavors of jobs requiring electric work. One of these constraints could be represented in English by “prefer a grade B electrician for a grade B job, use a grade A electrician only if no grade B electrician is available”.

Implementing IWM Scheduling for V-1: A Partitioning-Based Approach

Introduction

This section introduces the overall scheduling algorithm used in V -1. The algorithm first partitions the scheduling problem into a number of smaller scheduling problems, based on characteristics of the problem, such as location and skill. Each partition is scheduled by a different algorithm. The partitioning algorithm used in version -1 is described in this section. The next section describes the algorithm used to schedule each partition.

Partitioning the IWM Scheduling Problem: A Knowledge-Based Approach

The overall approach used in the version -1 scheduler is “divide and conquer”. The idea is that we use attributes of the problem, such as location and skill, to partition the set of jobs and the workers. We then have several smaller problems to solve. Assumedly, each of the smaller problems is easier to solve than the original larger problem. An example of a partition would be one responsible for scheduling all the electric jobs on a location to the electricians at that location.

For each partition, we can use a different algorithm to schedule the jobs in that partition to the workers in that partition. For example, we can try an exhaustive try-all-combinations approach. This might be the best algorithm to use if, for example, there are 1 or 2 workers and perhaps a couple jobs for, say, the plumbers at a location. Other algorithms may prove to be more appropriate for partitions with more jobs and workers.

Partitioning is a process of taking a problem space and dividing it into “regions”. In this case, the regions would be defined by the number of workers and the number of jobs for that region. An example might be all the electricians and all the electric jobs at one site. Partitioning results in the assignment of an algorithm

and a set of “parameters” to each region. These algorithms are then used to schedule each partition. The result of this activity is a set of schedules for all the jobs and all the work schedules.

Part of this process is a classification problem. This is the assignment of algorithms and parameters to each partition. In support of this effort, experiments could be run to determine how well various algorithms run under different parameterizations in different regions of the problem space. For example, for some of these algorithms, using different utility functions lead to different results. The utility function is thus one of the parameters that would be set for an algorithm, based on the characteristics of the problem space for that partition. The idea is to determine which parameter settings are useful in which regions for which algorithms.

Most of this experimentation will be done during the V – 1.5 development effort. In V – 1 the foundation will be built, and an algorithm will be assigned to each region. The V – 1 demo will illustrate the use of this technique, including the use of two different algorithms in different regions of the scheduling space. Of course, in the end, machine learning techniques such as decision trees² or Kahonen maps³ may prove useful in solving this classification problem. Alternatively, supervised learning techniques such as back propagation⁴ learning in neural networks may be used, by computing the utility of the schedules produced by each algorithm and defining the training set as pairs (input characteristics, algorithm and parameters with highest utility schedules).

The sub-field of AI known as distributed problem solving (DPS) considers knowledge-based approaches to task decomposition. One variant of DPS attempts to maximize a similar notion of utility. This work provides theory and examples that we may be able to leverage in future versions of the IWM scheduler. Partitioning as an example of distributed problem solving is further discussed in Appendix A.

IWM Scheduling Algorithms

Introduction

The Top Level Scheduling Algorithm for V – 1 is as follows:

Given:

A set of workers with their work schedules either empty (v – 1) or with jobs assigned (v – 1.5)
A set of jobs, some of which may have already been assigned to the above work schedules

Produce:

An assignment of jobs to work schedules such that
All the hard constraints are satisfied for each work Schedule
The soft constraints are optimized, based on the average goodness of fit of the schedules.

Algorithm:

1. Use the constraints on jobs being “admissible” for schedules to partition the problem space into regions.
2. Use TRE to determine the algorithm and parameters to be used to schedule work of each partition.
3. For each partition, use the TRE-designated algorithm to assign jobs to work schedules for that region.
4. Combine and persist results.

² A decision tree is a classification technique that uses the concept of information to determine which attribute contains the most information with respect to making a given decision. A decision node is then created, based on the possible values of this attribute, and the process is repeated.

³ A single layer neural network that performs unsupervised learning. An input is measured by its “distance” from each unit in the output layer, and the output unit closest to the input is pronounced the classification for the input. Then, the weights are automatically adjusted; that is, the system self-organizes over time. The network has the property that small deltas in input make for small deltas in output.

⁴ Back propagation networks are multi-layer Feedforward networks that uses the difference between the output nodes and the “true” value to adjust the connection weights between the network units.

This section considers the different algorithms used in step 2, including an exhaustive algorithm, probabilistic algorithms, and deterministic algorithms (Exhaustive Algorithms, Job Centric Algorithms, Work Schedule Centric Algorithms, and Unbiased Algorithms). One option in many of these algorithms is that of assigning a job to a work schedule in a process that includes making room for that job by pushing another job off the work schedule. Another optimization technique, known as pairwise swapping, is briefly considered.

Many of these algorithms make an initial pass at filling work schedules and then repeatedly traverse the unassigned jobs list, rescheduling to optimize overall utility. The relationship of this activity to scheduling unplanned work is then discussed.

All of the above solutions include the notion of an admissible work schedule, which is one where the location, building, skill, and time on shift of each job is compatible with the worker.

A basic concept used in many of these algorithms is the “goodness of fit” or utility of a work schedule or the set of all work schedules. This notion of “utility” is first discussed, as it influences many of the later considerations and design decisions. This includes the advantages of utilizing a fast, easily computed utility function which approximates the true (more complex to compute) utility function.

The Utility of Work Schedules

The notion of utility or “goodness of fit” is central to many of these algorithms. The basic idea is that some assignments of jobs to work schedules are more opportune than others are. This notion, which is known as utility, may be based on various attributes of the problem domain. For example, we may base utility on the priority and estimated time for the tasks on a job (the job’s utility), across all jobs on a work schedule (the work schedule’s utility), or across all work schedules (the total utility for all the jobs scheduled). We therefore talk about the utility of a job, of a work schedule and of all work schedules.

For V –1, the total utility of the set of work schedules may be computed through a linear computation. That is, the utility of the set of work schedules is equal to the sum of the utilities of each work schedule. Similarly, the utility of a work schedule is equal to the sum of the utilities for each of the jobs on the work schedule. These are not the only possible ways to combine individual utility functions (that is, the computations are not necessarily linear) but the linear functions are the simplest.

This is not the only way to compute the utility of a work schedule. Schemes that are more complex may be used. For example, the utility of a set of work schedules may be set to the minimum utility of any work schedule in the set. This would tend to minimize the variance in utilities between work schedules, making for more “even” work schedules, if not globally optimizing for the number of high-priority jobs processed.

A linear total utility function would seem to globally optimize the utility of each work schedule. Similarly, a linear global work schedule utility function tends to optimize the utility of each job on the work schedule. A linear work schedule and a linear overall utility function will therefore be used for V –1.

A straightforward family of functions, each of which satisfy the above requirements, consists of functions that sum each job’s utility for all jobs on a worker’s work schedule. The utility computation is thus reduced to computing the utility of each job. This may be determined by consideration of attributes of the job, such as priority and estimated time. One such utility function is called LC1.

LC1: An Example Utility Function based on Job Priority and Estimated Time

The simplest utility function of any interest in V –1 is based on a linear combination of an unweighted utility function (based on time on schedule) and a weighted utility function that also considers job priorities. This is the linear utility function LC1.

The LC1 utility function makes the following assumption:

- ❖ The utility of the set of work schedules is the sum of the utilities of each work schedule.

- ❖ The utility of a work schedule is a weighted sum of the unweighted and the weighted scores for the work schedule, as defined in the following.

First we define the time on shift, TOS, as the difference between the shift end time and start time:

$$TOS = WorkSchedule.ShiftEnd - WorkSchedule.ShiftStart$$

Then we define the job estimated time as the sum of the estimated time for all the tasks on the job:

$$job.EstimatedTime = \sum (\text{tasks in job}) \text{task.EstimatedTime}$$

The unweighted score for a work schedule, WSUnweighted, is defined as the percentage of the total time on shift:

$$WSUnweighted = \sum \text{over jobs} (job.EstimatedTime) / TOS$$

The weighted score for a work schedule, WSWeighted, is defined as the percentage of the total time on shift, where each job's Estimated time is weighted by the job's relative priority :

$$WSWeighted = \sum \text{over jobs} (job.priority \times job.EstimatedTime) / (job:maxPriority \times TOS)$$

The utility of a work schedule is then a linear combination of the unweighted and weighted scores:

$$WSUtility = (k)WSUnweighted + (1-k)WSWeighted \quad 0 \leq k \leq 1$$

By varying the value of k we can vary the relative importance of maximizing the priority of jobs on the schedule and minimizing free time on the schedule. For example, if k = 1 the utility only considers the total amount of unscheduled time on work schedules, while if k = 0 it penalizes the time given on work schedules to low priority jobs.

Our intention is to run experiments in version -1.5 that vary the value of k between 0 and 1 and computing the "quality" of the resultant schedules, for each combination of problem states. We will then pick a value of k that is optimal for each task to be assigned by the partitioning rules.

An analysis of the time complexity of LC1 is given in Appendix B.

Exhaustive Algorithm

The exhaustive algorithm tries every legal assignment of jobs to workers, returning those assignments that maximize the total utility of the set of work schedules. Results from the exhaustive algorithm are the benchmark against which the optimality of the results returned from any other algorithm is measured.

The exhaustive algorithm in V -1 is implemented as follows:

Set best score so far to 0

Repeat for each possible (admissible) assignment of jobs to workers (for that skill, location, and building)

 Create schedules for all workers

 Computer goodness of fit for the schedules

 If goodness of fit is greater than best score so far set best case so far to this case and best score so far to this score.

Return schedules produced by best case so far.

Note: This is computationally feasible for certain partition in V -1. The most complex location in the V -1 demo has 18 workers and will schedule about 120 of the jobs at this location. These are divided amongst the 4 skill types. The worst case is the maintenance technicians – there are 6 of them, so they might have approximately 40 of the jobs.

The combinatorics of matching 40 jobs to 6 workers are bounded by (a number much smaller than) 40 to the 6th power, which is 4,096,000,000 possibilities. For each possibility, we add time on shift and priority weighted time on shift, which might take approximately 20 operations (e.g., multiplications, see Appendix B), or about 80 billion multiplications, which should take a 400MHz server a few minutes. This estimate ignores the constraints related to time on shift (after the first few jobs are assigned to a worker there is no point in trying to add any more as there is no more time in the shift). It is therefore to large by a factor of between 2 and 10. The design goal of a two-minute deadline for scheduling 200 jobs may admit this solution in the case of small numbers of workers.

Note that this solution also provides the reference standard: The best set of schedules for the given workers and jobs is guaranteed to be found by an examination of every possible case.

This algorithm is limited, however, as it is approximately exponentially bounded by the number of workers for one skill at one site, which would limit its use in larger configurations.

The Rule set used in the first phase of Scheduling will assign the exhaustive algorithms to a region of the problem space for which it is determined to be appropriate. This determination is based on the number of workers and the number of jobs, independently of the details of the skill or location being scheduled.

Probabilistic Algorithms

Probabilistic algorithms repeatedly compute a schedule, based in part on the value of a random variable. An (approximately) optimal answer is derived by varying the value of the random value and using the value that maximizes the utility of the set of work schedules.

Probabilistic algorithms all follow the same general pattern:

- 2.1 Set best case so far = 0 and best score so far = 0
- 2.1 Repeat N time (e.g., N = 50)
 - 2.3 Create schedules for all workers, based on some random variable
 - 2.4 Computer goodness of fit for the schedules
 - 2.5 If average goodness of fit is greater than best score so far set best case so far to this case and best score so far to this score.
- 2.6 Return schedules produced by best case so far.

Note that this is an anytime algorithm⁵: at any point, steps 2.3-2.5 can be interrupted and the best case so far will be available.

The probabilistic algorithms vary in many dimensions. Some use the result of earlier steps as error indicators, basing the later algorithmic steps on a hill climbing or other search-like technique. A similar approach would be to cast the problem as one subject to simulated annealing. Other, more knowledge-based approaches might bound the admissible assignments by various domain-dependent factors.

The above are based on classic AI techniques, such as hill climbing. Other approaches come from the field of Distributed Artificial Intelligence (DAI). These include cooperative approaches. In these, jobs are initially (maybe randomly) assigned to workers. Pairwise swapping of jobs between workers is then used to optimize schedules.

There may also be benefit in attempted a distributed approaches based on the Contract Net. This would be done by conceptualizing each worker as an agent in a multi-agent system. The agents would bid on contracts let by the scheduler; each contract would assign a job from to a worker's work schedule. The basic loop would begin with the scheduler announcing the job. Worker objects would then bid on jobs, with

⁵ Anytime algorithms can be run for shorter or longer amounts of time, gaining in precision as time passes. Anytime algorithms provide more flexibility in real-time situations, compared with algorithms that run for a fixed amount of time regardless of context

the job being assigned to the highest bidder. A set of constraints would be used to control the interaction of worker agents as they offer to subcontract jobs from each other or to exchange jobs to optimize their schedules.

Some of these approaches can be combined, such as doing a weak search followed by pairwise swapping.

Deterministic Algorithms

Deterministic algorithms walk the Unassigned Job Queue or walk the set of work schedules, assigning jobs from the Unassigned Job Queue to work schedules.

Deterministic algorithms then optionally apply an optimization process known as rescheduling. This consists of walking the Unassigned Job Queue, attempting to assign high priority jobs to work schedules, possibly removing a lower priority job from the work schedule and placing it back on the Unassigned Job Queue.

Deterministic algorithms all follow the same general pattern:

- 2.1 Clear all work schedules, set Unassigned Job Queue to all jobs ready to be scheduled
- 2.2 Pick a direction to fill (job-centric, work schedule-centric or cross-product)
- 2.3 Fill work schedules from Unassigned Job Queue (with or without replacement)
- 2.4 Optional: Run rescheduling for the complete Unassigned Job Queue
- 2.5 Return work schedules and Unassigned Job queue

Note that there is a fixed time component to this algorithm (steps 2.1, 2.2, and 2.3). This is followed by the use of an anytime algorithm: at any point, step 2.4 can be interrupted and the best-case work schedules computed so far will be returned.

The degrees of variation of this series of algorithms, include:

- ❖ The focus of the algorithm: Walk the Unassigned Job Queue, walk the work schedule list, or consider both simultaneously
- ❖ Use of Replacement: whether one or more jobs may be removed from a work schedule to make room to add a new job.
- ❖ Use of Rescheduling: Once the initial pass at filling work schedules has been finished, whether the Unassigned Job queue is examined for possible work schedule optimizations.

Job-centric algorithms

Job-centric algorithms walk the Unassigned Job Queue, searching for a work schedule to put the job on, based on maximizing utility. For example, one algorithm examines all work schedules and adds the job at the top of the Unassigned Job queue to the work schedule that it adds the most delta utility to. This may or may not include replacement, which is the process of removing some of the jobs presently on the work schedule and returning them to the Unassigned Job Queue. The notion of utility of a work schedule is key to these algorithms.

This “local” approach to filling work schedules may be optionally followed by a more “global” attempt at optimization, provided by a rescheduling cycle or pairwise swapping.

These algorithms walk the Unassigned Job queue, examining each work schedule for each job element, computing the utility of adding that job to that work schedule. The time complexity of job-centric algorithms without replacement is therefore proportional to the product of the number of jobs, the number of work schedules, and the cost of computing the utility function.

Work schedule-centric algorithms

Work schedule-centric algorithms walk the list of work schedules: for each work schedule, the Unassigned Job queue is examined starting at the top. The top jobs that will fit are added to the work schedule, with or without replacement.

This “local” approach to filling work schedules may be optionally followed by a more “global” attempt at optimization, provided by a rescheduling cycle or pairwise swapping.

These algorithms walk the Unassigned Job queue for each work schedule. The time complexity of work schedule-centric algorithms without replacement is therefore proportional to the number of work schedules times the number of jobs.

Cross-product algorithms

Cross product algorithms consider the set of work schedules and the Unassigned Job Queue simultaneously. They compute the combination of job and work schedule such that adding that job to that work schedule adds greater delta utility than adding any other job to any work schedule, with or without replacement. This “local” approach to filling work schedules may be optionally followed by a more “global” attempt at optimization, provided by a rescheduling cycle or pairwise swapping.

Global Optimization: Rescheduling and Other Techniques

Introduction

Once the original scheduling pass has been completed, the work schedules and Unassigned Job queue are examined to determine if the total utility of the set of work schedules can be improved. Techniques used for this purpose include rescheduling and pairwise swapping, which are described in the following.

Rescheduling

Normally, scheduling is done at the beginning of shift for all the jobs in the Unassigned Job Queue. It may also be done ahead of time for the scheduled jobs for, e.g. each day for the next week.

Now consider what happens when an unplanned job is created. We must find a worker such that the job is admissible for that worker. It would be best if the worker has enough free time to add the job to its schedule. Barring that, a worker must be found that has a job on its Work Schedule that can be displaced by the new unplanned job. That is, a worker for whom the replacement leads to a set of work schedules with a higher total utility than without the replacement. In such a case, rescheduling is then called recursively on the displaced job.

This is defined by the constraints and preferences for replacing a job already on a worker’s schedule (call it the old job) with the unplanned job (call it the new job). Clearly, the work Schedule must be such that, if we remove the old job, there will be time in the shift for the new job.

The constraints are supplied by our concept of a job being admissible for a schedule. We rely on our fitness function to supply the above preferences. For example, it will insure that the priority of the new job is not lower than the priority of the job it will replace. It will also choose an old job to replace that optimizes the Work Schedule. That is, the resulting schedule has the highest priority of any schedule that could be made by replacing a job on the old work Schedule with the new job.

In general, we decide which old job to replace based on a comparison of the attributes of the old and new jobs, including their priorities and Estimated Times.

Example: An interesting rescheduling situation might feature a priority 4 job bumping a priority 3 job off a work Schedule, the priority 3 job in turn bumping a priority 2 job off another work Schedule. This complicated set of actions would occur if both of the following conditions were true. First, the priority 4 job has too great an Estimated Time to fit the work Schedule that includes the priority 2 job (not enough free time in the work Schedule even after the priority 2 job is removed). Second, the work Schedule that includes the priority 3 job will accommodate the priority 4 job once the priority 3 job is removed. It is also required that removing the priority 2 job from its work schedule will allow the priority 3 job to fit. The net result is that a priority 4 job has been added to the set of work Schedules for the shift at the cost returning a

priority 2 job to the Unassigned Job Queue. This results in an improvement in the overall utility for the complete set of work schedules.

Theory of Rescheduling

Introduction

Rescheduling may be defined algorithmically, based on the concept of work schedule and job utility.

Rescheduling is the process of moving high priority jobs from the Unassigned Job Queue onto a work schedule, with or without moving any jobs from the work schedule on to the Unassigned Job Queue (known as swapping).

Rescheduling is defined by a basic rescheduling loop, parameterized by:

- ❖ Rescheduling admissibility criteria: when is a job on the Unassigned Job Queue admissible to be added to a work schedule.
- ❖ Rescheduling halting criteria: when is a rescheduling loop terminated.
- ❖ Utility function: For a set of work schedules, a work schedule, and a job, as previously defined.
- ❖ Swapping policies: conditions under which jobs may be moved from a work schedule back to the Unassigned Job Queue during rescheduling.

This section begins with a description of the theory of rescheduling, based on the concept of rescheduling “time steps”. It then describes the basic rescheduling loop, including typical admissibility criteria, halting criteria, and swapping policies. It concludes with a description of the relationship between scheduling unplanned jobs and rescheduling.

Rescheduling Analysis via Rescheduling Time Steps

The rescheduling loops is quantified by the individual actions of adding a job to a work schedule, with or without replacement (swapping). We introduce the notion of a “time step”, denoted by s, t, u, \dots

We can now talk about, for example, the length of the Unassigned Job Queue at time t , or $JQLength(t)$. Similarly, assuming the Unassigned Job Queue is arranged from the job with highest utility to the job with lowest utility, then the largest utility of any job on the Unassigned Job Queue is $Util(JQ(1,t))$; that is, the job queue element with the greatest utility (ordered first in the list) at time t . Similarly, the smallest utility of any job on the Unassigned Job Queue is $Util(JQ(JQLength(t),t))$.

The above assumes that the utility function is not time dependent, an assumption made for $v-1$. This assumption is justified as “time” here means time within the rescheduling loop, all of which takes place in a few seconds.

The above analysis is used to determine sizing bounds for algorithms. Using this approach, theorems about the rescheduling algorithm may be proven, for example, by mathematical induction.

Basic Rescheduling Problem

Given at time t :

- The jobs on the Unassigned Job Queue ordered by utility from $JQ(1,t)$ to $JQ(JQLength(t),t)$
- The work schedules ordered by “work schedule order” from $WS(1,t)$ to $WS(NumWS(t),t)$

Produce at time $t+1$:

- The jobs on the Unassigned Job Queue ordered by utility from $JQ(1,t+1)$ to $JQ(JQLength(t+1),t+1)$
- The work schedules ordered by “work schedule order” from $WS(1,t+1)$ to $WS(NumWS(t+1),t+1)$

Such that:

The utility of the set of work schedules at time $t+1$ is the greatest possible, given:

- ❖ The set of work schedules at time t
- ❖ The admissibility, halting, and swapping policies
- ❖ The set of jobs on the Unassigned Job Queue at time t .

Note: "Work schedule order" may be by decreasing utility, random, or by some other domain-dependent metric. $\text{NumWS}(t)$ is the number of work schedules in the partition at time t . For $V = 1$ $\text{NumWS}(t)$ is a constant value, independent of t ; that is, the partitioning determines the number of work schedules in the partition, and this number does not change.

Basic Rescheduling Loop

Consider each job in the Unassigned Job Queue, call the typical one JOB_i

For JOB_i in $\text{JQ}(1,t)$ to $\text{JQ}(\text{JQLength}(t),t)$

Consider each work schedule, call the typical one WS_j

For WS_j in $\text{WS}(1,t)$ to $\text{WS}(\text{NumWS}(t),t)$

Let $\text{WS}' = \text{WS}(j,t+1)$ be the work schedule formed by adding JOB_i to WS_j with or without replacement

Let $\text{SWAPPED}(j,t)$ be the set of jobs swapped off WS_j at time step t (back to the Unassigned Job Queue)

If the utility of the set of work schedules after the replacement is greater than before, make the swap

If $\text{Util}(\text{WS}') > \text{Util}(\text{WS}_j)$
Add JOB_i to WS_j

If had to swap out any jobs add them to the Unassigned Job Queue, start again at the top of the Queue with the first work schedule

If $\text{SWAPPED}(j,t)$ not empty
Add each job in $\text{SWAPPED}(j,t)$ to Unassigned Job Queue
Set rescheduling job pointer i to 1
Set work schedule pointer j to 1

If we did not have to swap out any jobs then continue with next element in Unassigned Job Queue with the first work schedule

If $\text{SWAPPED}(j,t)$ empty
Set rescheduling job pointer i to $i+1$
Set work schedule pointer j to 1

If WS' utility not greater than WS utility then continue with next Work Schedule

If $\text{Util}(\text{WS}(j,t)) > \text{Util}(\text{WS}(j,t+1))$
Leave rescheduling job pointer at i
Set work schedule pointer j to $j+1$

Admissibility Criteria

To be admissible, a job must be "consistent" with the work schedule. This means all of the hard constraints must be satisfied. For $V = 1$ this includes location, possibly building, skill, and shift length (vs. job Estimated Time). All jobs on the Unassigned Job Queue are by definition consistent with the work schedule (or they would have been assigned to a different partition). To be admissible, a job must also fit on the work schedule. This is based on the Estimated Time for the job and for all jobs already on the work schedule, as compared to the Shift Start Time and Stop Time for the work schedule.

Admissibility criteria define the requirements for a job on the Unassigned Job Queue to be admissible to be added to a work schedule. The job must "fit" on the work schedule. Thus, the new work schedule, which includes the added job and does not include any jobs that are swapped out, consists of jobs whose sum estimated time is not greater than the time on shift.

Normally, a job will only fit if one or more other jobs are swapped out, although exceptions occur. A job on the Unassigned Job Queue is only admissible if there is a (possibly empty) replacement that leads to a work schedule with higher utility. That is, adding the unassigned job to the work schedule and then removing a (possibly empty) set of jobs from the work schedule results in a new schedule with higher utility than the old schedule.

Halting Criteria

The Unassigned Job Queue is ordered by utility. For example, in V-1 the Unassigned Job Queue is ordered based on priority and length of job, according to LC1. Then it must be true that any job in the Unassigned Job Queue of lower utility than an inadmissible job is itself inadmissible. Thus, once an inadmissible job is found in the Unassigned Job Queue no job to the “right” of that job need be tested.

$$\text{CanNotReschedule}(\text{JQ}(i,t)) \Rightarrow \text{CanNotReschedule}(\text{JQ}(j,t)) \quad j > i$$

Therefore the halting criteria at time t can be expressed by $\text{CanNotReschedule}(\text{JQ}(1,t))$.

Swapping Policies

Swapping policies define the types of replacements that can be done during rescheduling. That is, they define restrictions on which jobs can be removed from a work schedule and put back into the Unassigned Job Queue during the rescheduling pass..

Swapping policies define the number and characteristics of jobs that are admissible to be swapped out. For example, policies on the number of jobs that may be swapped out include (1) no swapping, (2) replace only one job, and (3) replace any number of jobs. They may also restrict jobs that may be swapped out to those under a certain priority or under or over a certain minimum or maximum estimated job time.

Swapping policies are also defined for some of the initial scheduling algorithms (e.g., the job-centric algorithms), and for stand-alone optimization procedures such as pairwise swapping (see below).

Scheduling Unplanned Jobs

As has been shown above, rescheduling may effect more than one job on the Unassigned Job Queue. It may effect each job on the Unassigned Job Queue as the rescheduling loop unfolds. Therefore, scheduling an unplanned job is not just a matter of fitting the job into a work schedule. Rather, it is a matter of fitting the job into the Unassigned Job Queue and then running rescheduling. This may lead to the new job being assigned to work schedules. However, based on the relative effects of the jobs on overall work schedule utility, it may lead instead to another job or jobs being assigned to a work schedule, rather than the newly created unplanned job.

Pairwise Swapping

Assume work schedules WS1 includes job J1 and WS2 includes job J2. It may be the case that swapping J1 and J2 either improves the total utility of the work schedules, or creates opportunities for further rescheduling to improve the total utility of the work schedules. This may happen, for example, due to non-linear total utility functions. It might also be due to the ability to create “holes” in work schedules that may be then used to add jobs from the Unassigned Job queue, jobs that would not have fit in either work schedule. This process (moving J1 to WS2 and J2 to WS1) is known as pairwise swapping.

Pairwise swapping is similar to techniques used in Genetic Programming wherein part of a “program” is swapped with another program; the new programs tend to survive if they have greater “fitness” than the old programs. The idea is to try different swaps and see which drive the fitness function to a relative maximum.

Pairwise swapping is not utilized in version -1.

The Future of IWM Scheduling

The original MasterLink IWM Development Plan was based around the use of a COTS job shop scheduler⁶. The idea was to wrap the ILOG scheduler as a CORBA component and then use it as the foundation for an IWM scheduler.

The use of a commercial off-the-shelf job shop-based scheduler may very well be the best solution for V 1.0. However, the results of running the version -1 scheduler on sample problems will determine whether it is worth attempting to extend the version -1 scheduler for version 1.0.

Another route would be to extend the approach taken in V - 1. From this viewpoint, the version -1 implementation would be the first step in a program of knowledge-based partitioning approaches to IWM scheduling. Future versions of the partitioning algorithm might use more knowledge of the tasks involved (perhaps a goal-based or planning approach). Scheduling algorithms might require more powerful scheduling heuristics or more complex utility functions.

There are definite advantages in using a COTS scheduler. Companies like ILOG have a large staff of people dedicated to advancing the state of the art of job shop scheduling, and we can leverage these efforts by supporting their latest offerings. This may require a smaller staff effort than designing, building and maintaining our own scheduler.

If, however, the partition-based approach proves applicable to environments dominated by soft constraints, there would be advantages in continuing the development of the IWM scheduler based on the knowledge-based partitioning approach.

IWM scheduling is an interesting domain, for which many approaches may be worth trying, including:

- ❖ Classic constraint-based scheduling, as in the ILOG scheduler.
- ❖ Knowledge-based task decomposition (partitioning) to decompose the scheduling into more tractable problems.
- ❖ Distributed problem solving based on the contract net or other protocols.
- ❖ Other approaches: learning approaches, neural networks, genetic algorithms, genetic programming.

Some of these approaches are more “researchy” than others. For example, a genetic approach may be based on treating each work schedule as a strand of DNA. Then the normal genetic algorithm mechanisms would be applied: work schedules would evolve with crossover and exchange of genes (jobs), the evolutionary “goodness of fit” being judged by the utility of the evolved set of work schedules.

A long-term research program might begin by attempting efforts utilizing several of the above mentioned approaches, and following the few that showed promise through second and third stages of development.

⁶For example the ILOG scheduler, which provides scheduling on top of an AI constraint-satisfaction package.

Appendix A: Partitioning as Distributed Problem Solving

Distributed Problem Solving (DPS) uses a divide-and-conquer approach to problem solving to reduce the complexity of the problem spaces considered. This provides solutions that require fewer workers. An issue in these types of solutions is that of task decomposition. In some cases, the designer builds the decomposition in⁷. Other examples of DPS involve knowledge-based approaches to task decompositions, similar in spirit to the V-1 IWM scheduler. These approaches are sometimes more sophisticated, in that they involve building explicit representations of the agent's beliefs, desires, and intentions (the so-called BDI architectures, an approach to practical reasoning that may be utilized in future IWM systems).

Task decomposition

Approaches to task decomposition approaches used in DSP include:

- ❖ Hierarchical planning. This is classic AI planning applied to the task decomposition realm.
- ❖ Taking advantage of hierarchical information intrinsic to the domain. For example, the task decomposition may mimic an AND-OR tree in the domain, such as is used to configure computers.
- ❖ Decomposing spatially, or based on information sources, decision points, or functionally. The V-1 IWM scheduler is an example of the later, decomposing on (location, skill) combinations.

Task assignment

DSP also includes approaches to assigning tasks to agents. This provides multi-agent solutions to decomposition problems. This may be of value as the IWM scheduler attacks more complex problem spaces. This would include spaces that place more reliance on soft constraints.

Task assignment algorithms used include:

- ❖ Table-driven assignment (fixed task types for each agent).
- ❖ Matching agents to tasks by mutual selection (like the stock market).
- ❖ Multi-agent planning.
- ❖ Contract net (see below).

Heuristics used to task assignment in DSP include:

- ❖ Matching tasks to agents by capabilities. This is similar to the way TRE is used to assign algorithms and parameters to the partitioned scheduling tasks.
- ❖ Avoiding overloading critical path resources.
- ❖ Having agents that are broad but not deep assign work to agents that are deep but not broad. An example in our system is the TRE-driven rules assigning work to the specific algorithms used to solve the partitioned scheduling tasks.
- ❖ Having a common or shared ontology between agents. This makes for solutions that are more coherent.
- ❖ Not reassigning tasks, except in cases of urgent tasks. An example of an urgent task in our domain is an unplanned job that the job manager has decided is ready for scheduling.

Example: The Contract Net

The contract net has been described⁸ as being most applicable to problems characterized by a well-defined hierarchy of tasks, a coarse-grained decomposition, and subtasks that either do not interact or interact positively when they do. The contract net protocol is used to organize a set of agents according to their capabilities and the tasks presented to them. It defines roles of "manager" and "worker", and proceeds with a bid and award cycle, as follows:

- ❖ The manager announces a new task.
- ❖ Some of the workers may choose to enter a bid for the task. The bid includes the agents capabilities and the workers required to satisfy the contract.
- ❖ The manager evaluates the bids and awards the contract.

⁷ This is not applicable in the case of IWM scheduling..

⁸ Michael N. Huhns and Munindar P. Singh, "Readings in Agents", Morgan Kaufmann Publishers, Inc., 1997.

- ❖ The worker carries out the contract.

In our case, we might build a system that, for example, represents each work schedule by an agent, and then let these agents bid on the jobs.

Appendix B: Analysis of the LC1 Utility Function

This section demonstrates an approach to quantifying the algorithmic time and space complexity of IWM scheduling utility functions. The idea is to provide a foundation for algorithms that compute the amount of time or space required to support the use of such utility functions, in support of the partitioning of the problem space into regions where different algorithms would be applied.

This section is highly speculative. In many senses it presents information in formalized terms that is intuitively obvious and usually expressed in less complex terms, including plain English. It was written in an attempt to inspire others to prepare a more formalized understanding of the domain.

This section is based on the LC1 utility function, described above.

LC1 is a linear combination of two terms, each of which contain a term for each job on the work schedule. LC1 may also be computed by taking the sum over all jobs of the unweighted and the weighted contribution for that job. That is, for a job J

$$\begin{aligned} \text{JobUtility}(J) = & k(\text{job.EstimatedTime /TOS}) \\ & + (1-k) (\text{job.priority/job::maxPriority}) (\text{job.EstimatedTime})/\text{TOS} \end{aligned}$$

The utility of the work schedule is then just the sum of the utilities of the jobs on the work schedules.

We can compute the computational complexity of this expression, by representing the work schedules and the jobs as vectors and taking the inner products to compute contributions, as in the following:

We define a vector of work schedules, from the first to the last in work schedule order (which may be random, decreasing with utility, or some other relationship). Thus at time t ,

$$W = \{W_1 \dots W_n\} \quad n = \text{numWS}(t)$$

We also define a vector of jobs, from the first to the last in order of utility

$$J = \{J_1 \dots J_m\} \quad m = \text{JQLength}(t)$$

These vectors can be used to define the job-centric, work-centric and cross-product algorithms mentioned above.

We define a vector of relative priorities, with one entry for each entry in J .

$$PRI = \{\text{Priority}(J_1)/\text{job::maxPriority}, \dots, \text{Priority}(J_m)/\text{job::maxPriority}\}$$

We also define a vector of relative job times, with one entry for each entry in J .

$$JOB_TIME = \{\text{Estimated Time}(J_1)/\text{TOS}, \dots, \text{Estimated Time}(J_m)/\text{TOS}\}$$

Finally, we introduce a unit vector, with one entry for each entry in J .

$$1 = \{1, \dots, 1\} \quad \text{length}(1) = m$$

Then the expression for the utility of the work schedule may be represented as

$$(k) \text{JOB_TIME} \cdot 1 + (1-k) \text{JOB_TIME} \cdot PRI$$

letting $r = 1-k$

$$\begin{aligned}
 & (1-r) \text{JOB_TIME} \cdot 1 + (r) \text{JOB_TIME} \cdot \text{PRI} \\
 & = \text{JOB_TIME} \cdot ((1-r) 1 + (r) \text{PRI}) \\
 & = \text{JOB_TIME} \cdot (1 + r(\text{PRI} - 1))
 \end{aligned}$$

or

$$\text{Job utility} = \text{JOB_TIME} \cdot (1 - (1-k)(1 - \text{PRI}))$$

The expression of job utility in this form expedites visualizing of the number of "operations" (e.g., multiplications) used in the computation of job utility. Working from the inside out

| | | |
|-----------|--|------------------------------|
| Computing | $1 - \text{PRI}$ | requires m subtractions |
| Computing | $(1-k)(1 - \text{PRI})$ | requires m multiplications |
| Computing | $(1 - (1-k)(1 - \text{PRI}))$ | requires m subtractions |
| Computing | $\text{JOB_TIME} \cdot (1 - (1-k)(1 - \text{PRI}))$ | requires m multiplications |

Thus, the computation of one job utility under LC1 requires $2m$ multiplications and $2m$ subtractions, or $4m$ operations.